

## EE49 Laboratory 1

### Source Coding

Due Date: 4/15/2011 at 3pm in Lab

After completing this lab assignment, you must be checked off by a TA in lab and provide your code via email. If you cannot make the 1-3pm Friday timeslot, it is your responsibility to setup an alternate appointment with the TAs before the due date. Good luck!

**GOAL:** The goal of this lab is to gain an understanding of the different components involved in source coding. Specifically we will be dealing with the Discrete Cosine Transform (DCT), sample quantization, and Huffman coding

**Note:** For your use, a few subVIs like *binary\_to\_decimal.vi* and *decimal\_to\_binary.vi* are provided.

## 1 Introduction to Source Coding

---

Open "lab1.vi" and try to understand the overall data flow involved in source coding. The code can be looked at as roughly consisting of 8 blocks where each block performs a dedicated function. You will be responsible for the "encoding" portion of this lab. The decoding portion is completed for you. These blocks are:

### (1) Reading the audio file

Lab VIEW has a customized block which reads a raw audio file (.wav) and outputs the audio waveform. In fact, it outputs an array of waveforms (in case of stereo, we get 2 waveforms one each for the left and right speakers). In our case, since the .wav file has only 1 channel, we extract it directly. Note that, unlike Matlab, the numbering to elements in an array starts from 0 in LabVIEW. Each waveform has an array of time samples (denoted by "Y") and the sampling time (denoted by "dt"). It is this array of time samples that we wish to communicate efficiently.

### (2) Performing DCT and throwing away some of the DCT coefficients

Our next goal is to try and compress this array of time samples. One standard technique for compression uses a frequency domain representation of the signal. Typically, an audio signal is mainly contained in the lower frequency range. Hence, in order to compress the signal further, it is common practice to ignore the higher frequency components (round them to 0). Here, we make use of the Discrete Cosine Transform (DCT) to obtain a frequency domain representation of our audio signal.

### (3) Quantize DCT coefficients

The remaining DCT coefficients from the previous step are real numbers. Since digital communication uses binary communication, we cannot transmit these real numbers with full precision. There is a trade-off between the amount of data we need to transmit versus the precision we require. "Resolution" is the number of bits we will use to express each DCT sample. Thus, if we set the resolution to 4 bits/sample, then we would need to transmit only 4 bits of data per DCT sample. However, we would be able to represent only  $2^4 = 16$  different DCT values and all other values would have to be rounded off to one of these 16. This stage performs the required quantization according to the "Resolution" you choose.

### (4) Huffman Encoding

Following quantization, we are left with an array of frequency-domain samples where each sample is a binary number of length "Resolution". In this array, some of the samples will be repeated more often than others. In order to exploit this redundancy and compress the data further, we will make use of Huffman encoding as discussed in class.

### (5) Huffman decoding

This is the first stage on the receiver side where the Huffman code is decoded (note that the receiver will also need to know the parameters of the Huffman code in order to decode it).

### (6) Dequantize

The array of binary numbers obtained after the previous stage are converted back to an array of reals at this stage.

### (7) Zero padding followed by Inverse DCT

The reals obtained in the previous stage are the receiver's estimates of the DCT coefficients. Note however, that some DCT coefficients were thrown away before quantization. In this stage we replace all the rejected DCT coefficients by 0 and perform an Inverse DCT on the received estimates.

### (8) Forming the reconstructed waveform

Finally, we form a waveform with the time samples obtained from the Inverse DCT and play it back.

For this lab, you will be responsible for building parts of the huffman encoder. The decoder block is done for you.

## 2 Finish the DCT block

---

Firstly we wish to write a DCT block. It should accept an array of double precision reals and output an equal sized array of DCT coefficients (double precision reals again). We will be using the standard formulae for DCT calculation:

$$X_k = \begin{cases} \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n & \text{if } k = 0 \\ \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} \left( x_n \cos \left( \frac{\pi(n + \frac{1}{2})k}{N} \right) \right) & \text{if } k \neq 0 \end{cases}$$

where  $x_0$  to  $x_{N-1}$  is the input array and  $X_0$  to  $X_{N-1}$  is the DCT of the input array. Note: Depending upon how you implement the DCT operation, it may take some time to run.

**Write the logic block in *DCT\_block.vi***

In order to test your code, use the *DCT\_block\_test.vi*, enter the audio.wav file present in your folder as input. The DCT of the file should look like the one shown in 1.

## 3 Quantize to 2D array of bits

---

We need to quantize an array of reals to an array of binary numbers ( where sample length = "resolution" bits), or equivalently a 2 dimensional array of bits (where each row represents one sample). For this purpose, we use a notation called the offset binary. The algorithm is as follows:

- Find the "full scale" of the array, where "full scale" is defined such that all elements of the array lie between  $-\frac{\text{full scale}}{2}$  and  $\frac{\text{full scale}}{2}$ .
- If the "Resolution" is set to  $L$  bits, the interval  $[-\frac{\text{full scale}}{2}, \frac{\text{full scale}}{2}]$  needs to be divided into  $2^L$  intervals. Thus the interval size can be computed as  $\frac{\text{full scale}}{2^L}$ .
- Finally, the offset binary representation of a real number ' $X$ ' is defined as

$$\text{Binary} \left\{ \frac{X + \frac{\text{full scale}}{2}}{\text{interval size}} \right\} \quad (1)$$

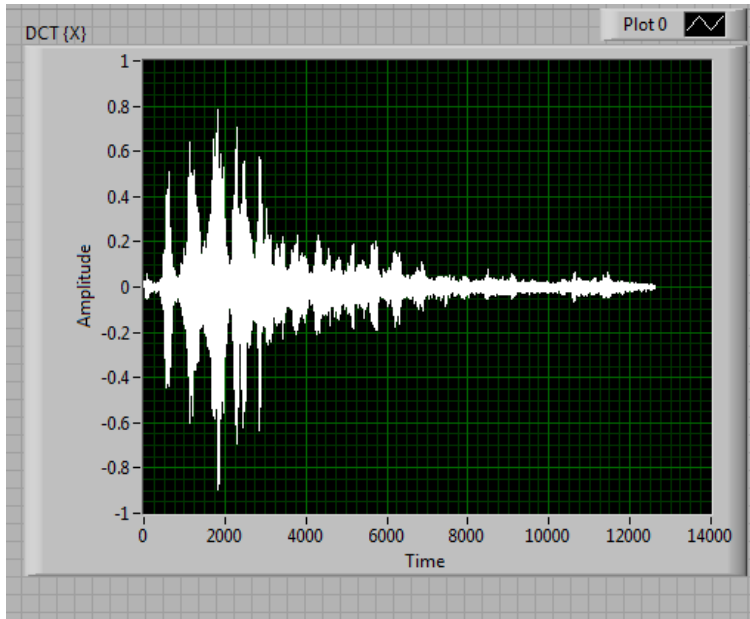


Figure 1: DCT of audio.wav

where  $Binary\{x\}$  is the representation of  $x$  in the binary format. The  $Binary\{x\}$  function is provided for you as *decimal\_to\_binary.vi*. You must first create the offset representation and then pass it through the provided function.

Note here that we use MSB first notation, i.e. the most significant bit for each sample is the  $0^{th}$  element.

**Write the logic block to quantize the array of real numbers into a 2D array of bits in *Quantizeto2Darray.vi*.**

In order to test your code, enter the input as shown in 2. The output of your *Quantizeto2Darray.vi* code should look like the output shown in 2.

## 4 Forming the Huffman Tree

Coming to the final stage of source coding, we need to perform Huffman coding on the array of binary numbers obtained above. Note that each of these binary numbers will be of length  $L$  (where  $L$  = the resolution you choose on the front panel of *lab1.vi*). The sub-VI, *form\_frequency\_distribution.vi* takes the 2-D array of bits and counts the number of occurrences of each of the binary numbers, thus forming a frequency distribution table.

The next stage of the Huffman encoding requires us to form the Huffman tree. We store this tree as a cluster consisting of 6 arrays, namely,

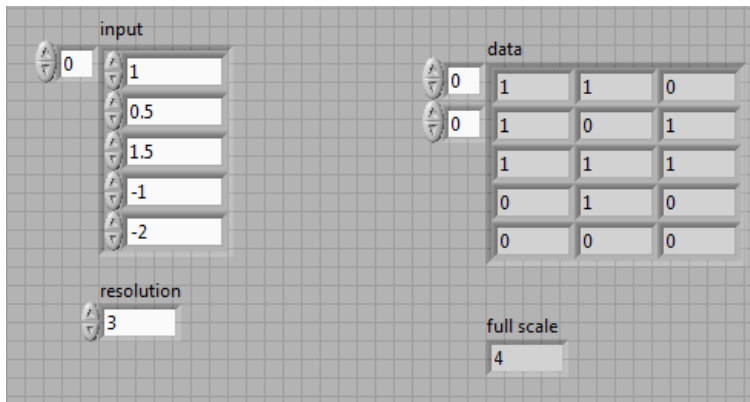
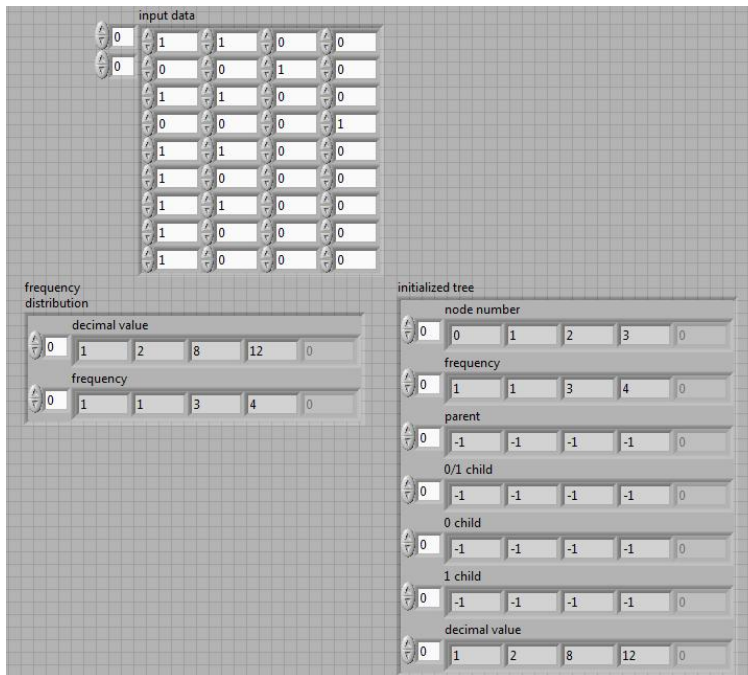


Figure 2: Output of Quantizeto2Darray.vi

- Node number: Each node in the tree is given a number (starting at 0).
- Frequency: Occurrence frequency of the binary number corresponding to the node (for non leaf nodes, it is the sum of the frequencies of its two children).
- Parent: Number of the parent node (defaults to -1 if the node has no parent or the parent is still unknown).
- 0/1 child: Let's use a notation where we call the children of a node as the 0th child and the 1st child. This is an array of 0s/1s representing whether the node is the 0th child of its parent or the 1st child. (defaults to -1 if the node has no parent or the parent is still unknown).
- 0 child: The number of the 0th child of the node (defaults to -1 if the node has no children).
- 1 child: The number of the 1st child of the node (defaults to -1 if the node has no children).
- Decimal value: The decimal value of the binary number corresponding to the node.

The sub-VI *form\_frequency\_distribution.vi* outputs an initialized version of this tree where only the leaf nodes are accounted for. **You must add entries to the tree cluster for each of the parent nodes.**

**Write the logic to build the full tree from this initialized version in *form\_tree.vi*.** To help you visualize the exact data formats and tree-creation process, let's discuss a small example.

Figure 3: Output of `form_frequency_distribution.vi`

Firstly, let's say that the array of binary numbers is as shown in the "input data" field in figure 3. The figure shows the output obtained after running the `form_frequency_distribution.vi` on this input data. Note here that the binary sequences "1100" and "1000" appear 4 and 3 times respectively and the binary sequences "0010" and "0001" appear once each. The frequency table is shown in figure 3, in the included tree cluster, under the "frequency" label. The decimal representations of "1100" and "1000" are "12" and "8" respectively and for "0010" and "0001" are "1" and "2" respectively. The Huffman tree for such an array is shown in figure 4 below.

Note that the initialized tree accounts for only the 4 leaf nodes. The final tree (output of `form_tree.vi`) should however look like as shown in 5.

## 5 Form the codebook

We now have the Huffman tree ready, using this we need to generate the codebook. The 'codebook' is a mapping from the binary numbers (of resolution 'L') to the Huffman sequences. It's like a dictionary where you can look up any L-bit binary number and find its corresponding Huffman sequence. For creating this dictionary, we need to find the Huffman encodings of all L-bit binary

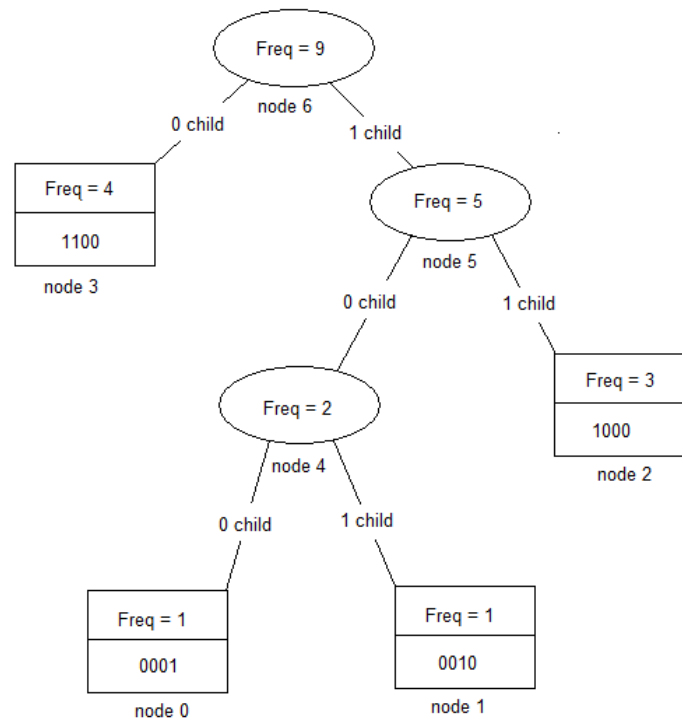


Figure 4: The tree

numbers which have a non-zero frequency. Remember from class that to get this Huffman encoding, we will need to traverse the tree upwards (from a leaf node to the root) and get the appropriate huffman code bit sequence from this path.

For eg. refer to the figures 4 and 6, the decimal value '12' is a 0 child of the root node and hence the huffman representation of '12' (i.e. "1100") is "0". Similarly, the decimal value '1' (i.e. "0001") is the 0-child of node 4, which is the 0-child of node 5, which itself is the 1-child of the root node and hence the huffman representation of '1' (i.e. "0001") is "100". Make sure you understand this algorithm well.

Eventually, we wish to output 2 arrays, one array of decimal values (of the leaf nodes) and another array of strings (the corresponding Huffman codes of the leaf nodes).

## 6 Finishing off the Huffman encoder

Once the codebook is formed, encoding an incoming bitstream is just a lookup from the 'dictionary' and this part has been given to you. If the two blocks described above (*form\_tree* and *form\_codebook*) work for you, then this should work too. However, one final check can be made by entering the inputs as shown

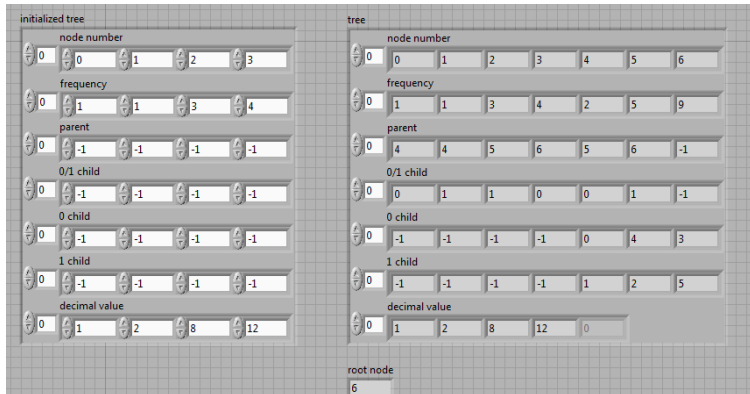


Figure 5: Output of form\_tree.vi

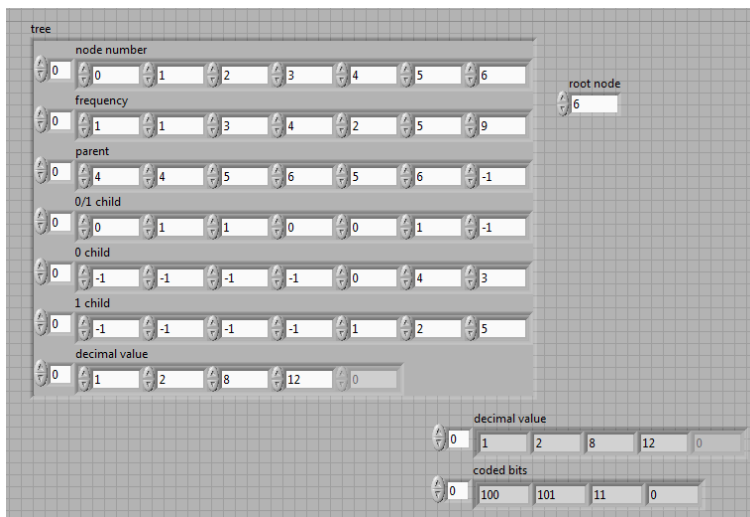


Figure 6: Output of form\_codebook

## 7 Putting it together

Play around with lab1.vi after solving all the above exercises. Enter the filename of the audio file in the "path" control and enter resolution bits. You can adjust the two yellow cursors on the DCT waveform graph to choose which DCT coefficients you would like to preserve. Try out audio.wav file present in your folder. Feel free to try out any .wav files you find and report interesting observations! Good luck!

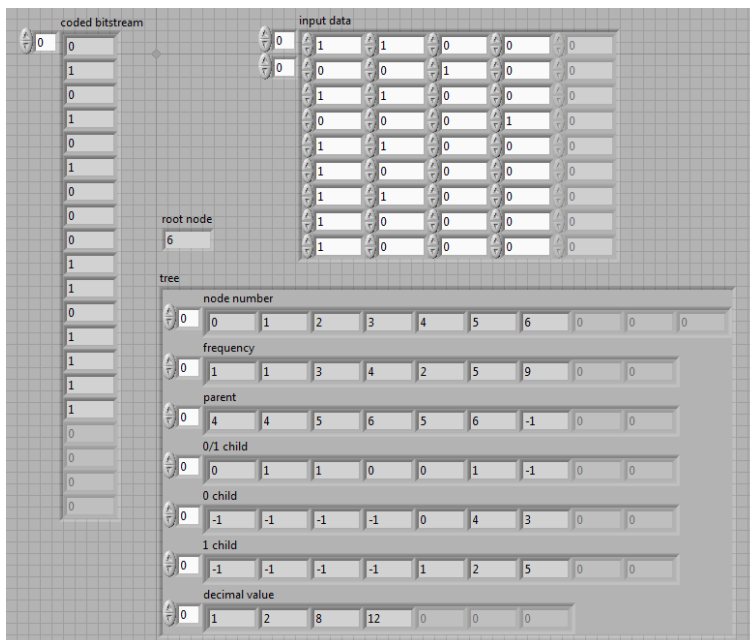


Figure 7: Output of huffman\_encoder